MAST
MEGA-AMPERE SPHERICAL TOKAMAK

IDAM

D.G.Muir February 2010

Accessing Arbitrary Hierarchical Data

Accessing experimental data is relatively straightforward when data are regular and can be modelled using fixed size arrays of an atomic data type, e.g., a time series measurement of type double.

It is more difficult however when data are structured hierarchically and when the hierarchy can change on a variety of (human) timescales: days, weeks, months, years.

Examples of this are the meta data associated with MAST and its numerous diagnostic configurations. These are shot range dependent – shot being a surrogate for date. Also, some types of meta data may be time dependant, e.g., positions that change, either because of thermal expansion or mechanical movement, during a shot.

Our experience of handling this type of data is limited: efit++ uses hierarchical descriptions of coil positions encoded in XML. The ITM (with our assistance) are busy collecting MAST machine descriptions and data mappings, also encoded as XML.

IDAM

Both these systems are flawed, however, as they rely on pre-defined data structures. These can only change on a slow human timescale. They are either defined within hand-crafted code (as was done for efit++) or created automatically using software tools (ITM uses XSLT with XML Schema, MAST has experimented with gSOAP). In all these cases, new client libraries must be issued to propagate changes to structures. Otherwise application codes will break.

Neither approach can handle dynamically generated hierarchical data where the hierarchy is unknown, or where the data structures can change on fast timescales.

Neither approach can accommodate the needs of different groups of users who require to use the same analysis models and data access tools with hierarchical data that differs in organisation.

Either we adopt a single standard and accept the imposed limitations or we embrace an alternative approach where we don't need to know how data are organised when our codes are built – just how to access them.

This way codes access data in the same way we access data. When we run a query against a database or list the contents of a hierarchical file system, we don't know how or what data will be returned. That however doesn't stop us from using the information provided.

Available Technologies

The problem of passing hierarchical data between a client and a server is not new. There are many technologies that will perform this task, e.g., COM,DCOM,CORBA,ORB,ICE,…

In essence, structured data is decomposed into primitive data types (un-marshalling or de-serialising), these are passed across a network using a middleware technology with a data interface defined using an interface definition language (IDL). The client then reassembles (marshals or serialises) the structures.

Both the technologies and the IDL are capable of significantly more complex tasks than just transporting hierarchical data structures.

An IDL is a language neutral means of joining distributed system components. It generates a data binding between client and server components.

IDAM needs an IDL that generates the interface dynamically at runtime. There must be no fixed data binding. This rules out these standard technologies.

IDAM

Developments within IDAM have created a new component that does the following:

- Passes hierarchical data of known structural types, as used by efit++ and the ITM group.
  - e.g. from XML using a SAX parser with pre-defined data structures.
  - e.g. from XML using gSOAP and XML Schema: Data bindings create the structures.

- Passes hierarchical data of un-known structural types:
  - Created dynamically from netCDF-4 or HDF5 when data are encoded as user defined data structures.
    - this capability is a requirement of users needing to access data from the latest version of efit++.
  - Created dynamically from public XML schemas and XML instances
    - e.g., enables ITM schemas to be used (not tested)
  - Created dynamically from private XML schemas and XML instances
    - enables private data compliant with private schemas to be used (not developed)
  - Created dynamically from the results of SQL queries against a database
    - a requirement of collaborators from YORK who need to query the MAST CPF
  - Passes netCDF4 or HDF5 data as group rooted sub-trees – data access is at the group level, not just at the variable level.

- No need to install new client libraries when structures change: its all automatic.
- Bindings to IDL, Fortran, C, C++, …

MAST
MEGA-AMPERE SPHERICAL TOKAMAK

IDAM

- ➢ Thin library: no additional libraries for technology components required. All processing is server side.
- ➢ Library of tools to discover what data are available and to access them.
- ➢ System architecture independent: 32 or 64 bit, big or little endian. (Linux only currently)
- ➢ Parallel processing friendly (MPI? openMP? Not tested)
- ➢ has the potential of enabling data to be mapped between different data XML schemas. (not tested)

How does it work?

Consider three general problems:

1. Passing fixed definition data structures
2. Passing structures defined using XML Schema
3. Passing structures defined only when data are read, e.g. from netCDF-4 files, that are not known beforehand.

And consider how these passed data can be accessed using the same toolset.

Passing fixed definition data structures

An example of a fixed definition data structure defined in the c programming language is as follows:

```
struct DATA {
   int  nco;                         // Number of Coordinates/Elements
   float *r;                         // Radial Position
   float z[100]                      // Z Position
   USERSTRUCT x[10];                 // User defined data type (structure)
} ;
```

Data organised into structures occupy a chunk of memory with a fixed size. Components of a structure have a position within this chunk that depends on the other components that occupy space before it: what data types they are and what their sizes are. For example, in the above data structure, the integer component ***nco*** is positioned at the start of the memory. It has a position offset of zero. It also has a size of 4 bytes. The next component is a pointer to an array in heap memory. This has a positional offset of 4 bytes. It has a size that depends on the system architecture - 4 bytes on a 32 bit machine and 8 bytes on a 64 bit machine. The next item is a fixed length array of type float. Its position depends on the system architecture because of the pointer that precedes it. Its size is 400 bytes. Last is a user defined type – that may be another structure.

Data organisation within data structures also depends on data packing factors. These relate to how data are fetched from memory efficiently. Consequently, there may be packing bytes between structure components.

Passing fixed definition data structures

So why don’t we just send the chunk of memory as a set of bytes from the IDAM server to the client?

We could if the system architectures were identical and there were no pointer types within the structures. This would be very limiting however – most data are not like this and system architectures are varied.

Pointers mean that when structures are passed, local system architecture changes the structure size. Also, the actual value held by the pointer on the server is meaningless on the client – it will not point to the data. So the data that the pointer points to has also to be sent and in a way that ensures the correct values are stored.

Another difference between architectures is endianness – the ordering of bytes within 16, 32 and 64 bit words: most or least significant byte first - big and little endian respectively. (The term comes from Swift’s Gulliver’s travels and refers to cracking open hard boiled eggs at the small or big ends!) So a stream of bytes to a client with a different endianness will reverse the byte ordering.

In order to pass fixed data structures, we need to know the order of structure components, their type, position and size. We also need to know what data they point to, how many items – rank and shape. And we need a means of passing data taking into account architecture differences: endianness, pointer sizes, structure alignment packing.

Passing fixed definition data structures

How does it work in practice?

Given a set of fixed definition data structures, the IDAM server parses the definitions into structure components and records the properties of each component: type, size, pointer, etc.

When the server is ready to pass a structure, it sends the details of the structure definition across a data interface known to both client and server. These details define a new data interface so that the client now knows what is being sent and in what order: name, type, size, pointer, rank, shape, etc. The client is aware of its own architecture so can modify the structure definition according to pointer size and packing requirements.

When the server passes the data, using XDR middleware that corrects for endianness, the client allocates memory for the data and writes into this memory at appropriate locations all the data components as they arrive.

Hierarchical data are managed by making this procedure recursive.

Many different fixed definition structure types can be passed in this manner, e.g., the efit++ meta data structures or the ITM CPO data structures.

The definition parse step can be avoided by storing the parser output in the IDAM database. A simple database query would then return the definitions required.

MAST MEGA-AMPERE SPHERICAL TOKAMAK

IDAM

Passing structures defined using XML Schema

An example of a data structure defined by XML Schema is below:

```
<xs:complexType name="listFloatType">
  <xs:simpleContent>
    <xs:extension base="floatList">
      <xs:attribute name="units"                         type="xs:string“ default="m"/>
      <xs:attribute name="delimiter"                     type="xs:string“ default=","/>
      <xs:attribute name="vector"                        type="xs:float“ />
     <xs:attribute name="sizeVector"                     type="xs:int"  default="0"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

This is used by data binding compilers like gSOAP to create data structure definitions (below) and the software required to create instances of data structures by parsing XML instances of the schema.

```
struct ns1__listFloatType {
  char *__item;
   char *units;
   char *delimiter;
   float *vector;
   int sizeVector;
};
```

MAST MEGA-AMPERE SPHERICAL TOKAMAK

IDAM

Passing structures defined using XML Schema

The method employed to dispatch these types of data structure from the server to a client is identical to that for fixed definition data structures.

The only difference is in the way the definitions are created and instances generated: by hand coding or automatic coding by gSOAP.

Several different XML Schemas could be known to the server at compile time so that the various definitions are fixed and the set of parser software is included in the build. XML instances could be accessed from the IDAM database.

Alternatively, private XML schemas and XML instances could be passed to the server from a client and the server could create private structure definitions, the parser software, and the means of running the parser and accessing the data structures – all automatically. These structures could then be passed back to the client.

This approach is not simple. gSOAP does not have the string stream functionality required to process local XML documents – it is a client server system and expects the XML from a socket based server using tcp/ip. For this to work with IDAM gSOAP was modified. This creates a maintenance issue best avoided.

Passing structures defined only when data are read

This is the most pragmatic and powerful approach to both recording hierarchical meta data and passing hierarchical data structures.

Advanced file systems such as netCDF-4 and HDF-5 allow users to define their own data structure types and to record data using these types. These structures can be hierarchical in that components may themselves be a user defined type.

When data are accessed from these file systems, the definitions of the user defined types are made available. These definitions are easily mapped to the structure definitions used by IDAM. Instances of data can then be sent to an IDAM client in exactly the same way as before.

There are 4 types of user defined data structure:

Compound:	similar to normal data structures
Variable Length:	an array of arrays of different length (and type)
Opaque:	a blob of binary data – passed without interpretation.
Enumeration:	a set of integer values that have specific meanings (labels).

In addition to user defined types, any sub-tree drawn from the data hierarchy can be mapped to a set of structure definitions and the whole data tree sent to the IDAM client.

How can passed data be accessed

Assume data is available to a client following a request for hierarchical data. Assume also the user does not know the contents of the data.

- The first step must be to ask what's available.
- The second step is to ask for data.

How is this best accomplished.

Hierarchical data is best represented as a data tree. The root tree node is the starting point and from this node there are multiple branches (children). Each of these child nodes is a root node of a sub-data-tree and may have other branches. Nodes without children are terminating or leaf nodes.

Hierarchical data from the server are stored on the client as a data tree. Each tree node has an associated component of data with a type.

To ask what's available is to request a list of the contents of all tree nodes.

To ask for data is to identify a specific tree node and to obtain the data recorded there.

The IDAM toolset provided functions to both query and access what's there.

For IDL applications, all this is done automatically: IDAM returns a data structure for you.

How can passed data be accessed

There are two ways we can work with the data returned from the IDAM server.

When we know the structure type, we can cast the data to this type and so work with the data using knowledge of its contents. This is trivial and will be not be discussed further.

When we don’t know the structure, we have to do some investigating first before we can access the data. Once we know what’s there, we can access the data.

The IDAM API has the form: **handle = idamGetAPI(data, source)**

Where **data** is a string identifying a data object or signal, and **source** is a string identifying a data file or a shot number or some other identifier known to the IDAM database. The API returns an integer **handle** that’s a simple index into an array of returned data structures. This value is used in all subsequent IDAM function calls.

The first action is to register the data tree we’re interested in using the handle value returned by the API.

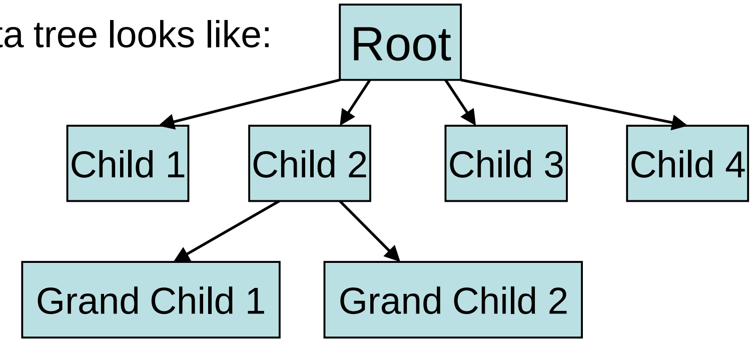
rc = setIdamDataTree(handle);

The return code, rc, takes two Boolean values: 0 (False) or 1 (True). These indicate whether or not the data referred to via the handle value is a hierarchical data tree. Regular signal based data returned by IDAM are not hierarchical. Best practice is to test this returned value.

Thereafter, the rich set of accessor functions can be used to explore the data tree, to search for specific data items and to return data.

How can passed data be accessed

A simple data tree looks like:



Each tree node has a name and contains data. These data items are structures with a Type Definition. These structures contain other structures (attached to child nodes) and atomic data elements.

To explore the tree, the first step might be to print it. A number of functions exist to print various aspects of a tree. The simplest is ***printNTree***. This displays the name of each node, the number of children (branches) and the node's data structure definition table.

This should be sufficient to identify all the key information required to identify and access data from the tree.

The next step is to locate specific data within the tree. This data can be targeted in a number of ways: the name of the tree node; the structure definition name; the name of a structural element; the name of a structural element's structure definition.

How can passed data be accessed

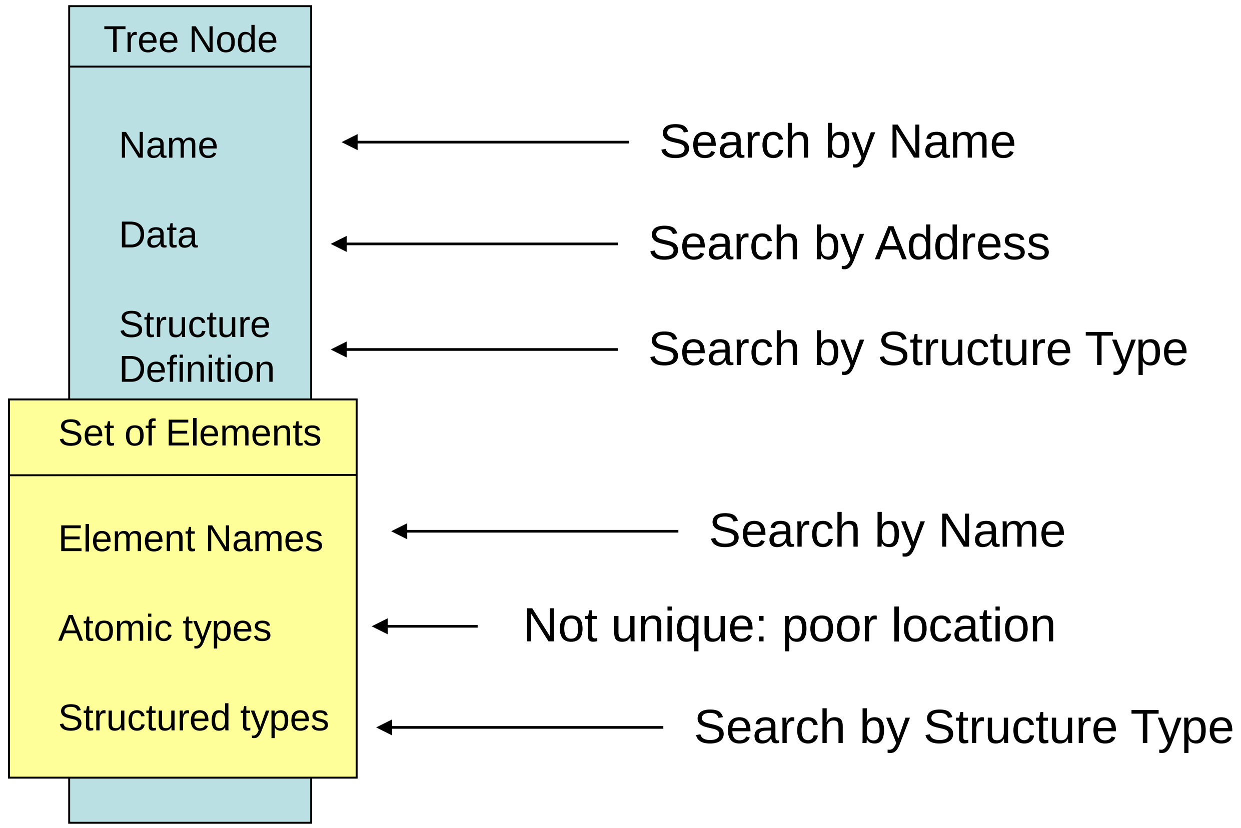
A tree Node contains two structures: the data and the data's structure definition.

Searching means locating data using name or type information.

Names are hierarchical so data can be located using the naming syntax: a/b/c/d.

Once a data node has been found, other functions are used to extract the data's properties: name, type, count, size, rank, shape and description.

Finally, all data eventually deconstructs to atomic types. These can then be accessed.



How can passed data be accessed

The simplest method of demonstration is with IDL:

```
handle = idamgetapi(data, source)                  ; Access the data and create a data tree

istree = setidamdatatree(handle)            ; Register and check the data is a hierarchical data tree

if(istree) then begin
    node = getidamnodechild(handle, 0, 0)          ; Data is always located in the first child node
    struct = makeidamstructure(handle, node)       ; Create a data structure from the tree
endif
```

All functions, apart from makeidamstructure, are part of the IDAM DLM library.

The makeidamstructure function is recursive: it extracts information from the data tree and builds a complete hierarchical data entity. How it operates demonstrates clearly how data are extracted from the data tree.

How can passed data be accessed

A simplified version of the makeidamstructure function (no checks or error handling) is:

```
function makeidamstructure, handle, tree

  forward_function makeidamstructureitem

  parent = getidamnodeparent(handle, tree)
  childid = getidamnodechildid(handle, parent, tree)                ; Base Branch ID

  count = getidamnodestructuredatacount(handle, ntree)              ; Count of Tree Node Structure Array elements

  str = makeidamstructureitem(handle, ntree)                        ; First structure array item

  for j=1, count-1 do begin
    node = getidamnodechild(handle, parent, childid + j)
    nstr = makeidamstructureitem(handle, node)                      ; Build Structure Hierarchy
    if(is_structure(nstr)) then str = [str, nstr]                   ; Build Array of Structures
  endfor

  return, str
end
```

This function calls the makeidamstructureitem function

How can passed data be accessed

A simplified version of the makeidamstructureitem function (no checks or error handling) is:

```
function makeidamstructureitem, handle, tree
  forward_function makeidamstructure
  acount    = getidamnodeatomiccount(handle, tree)                    ; Count of the Tree Node Structure's atomic type components
  anamelist = getidamnodeatomicnames(handle, tree)                    ; Names of the Atomic Components
  apointer  = getidamnodeatomicpointers(handle, tree)                 ; Is this Atomic Component a Pointer ?
  if(acount gt 0) then begin
    for j=0, acount-1 do begin
      data = getidamnodeatomicdata(handle, tree, anamelist[j])        ; Get atomic data cast to correct type
      if(apointer[j] and is_number(data)) then begin
        p=ptr_new(data)
        if(j gt 0) then astr = CREATE_STRUCT(astr, anamelist[j], p) else astr = CREATE_STRUCT(anamelist[0], p)
      endif else begin
        if(j gt 0) then astr = CREATE_STRUCT(astr, anamelist[j], data) else astr = CREATE_STRUCT(anamelist[0], data)
      endelse
    endfor
  endif
  scount    = getidamnodestructurecount(handle, tree)         ; Count of the Tree Node Structure's structure type components
  snamelist = getidamnodestructurenames(handle, tree)         ; Names of the Structured components
  if(scount gt 0) then begin
    for j=0, scount-1 do begin
      node = findidamtreestructure(handle, tree, snamelist[j])
      if(j gt 0) then sstr = CREATE_STRUCT(sstr, snamelist[j], makeidamstructure(handle, node)) else $
        sstr = CREATE_STRUCT(snamelist[0], makeidamstructure(handle, node))
    endfor
  endif
if(scount eq 0) then return, astr                 ; If no structured component, then return the atomic elements
if(acount eq 0) then return, sstr                 ; If no atomic component, then return the structured elements
return, CREATE_STRUCT(astr, sstr)                 ; Combine both Atomic and Structured components
end
```

Examples

MAST shot date and time

MAST data sources

MAST signals

EFIT++ Meta Data

MAST Limiter positions

EFIT++ user defined data item

EFIT++ sub-tree